Prototype Design Specifications

VR Casino: Simulated Virtual Reality for modelling financial risks

Developers: Elvis Chen, Emaad Fazal, Tobi Onireti, Aleem Haq

Introduction

The purpose of this Documentation is to provide detail on the Software Design of the Prototype for the Casino VR Research Project. Within this prototype shows basic implementation of the main functions for the final project. This prototype contains a working environment, slot machine and teleportation from endpoint to endpoint within the environment. This prototype contains a functional slot machine that follows the NearMisses theory based on Holt and Laury Risk Aversion.

Programming Interface

Unity

The prototype is designed using Unity. Unityis used to render the virtual reality environment and all programming involving any sort of user interaction within the environment. For the backend scripting, it was coded in C# and it is used to create the scripting functionality to follow the requirements from the Holt and Laury Risk Aversion : Research Paper. C# will be used to generate scripts for the individual objects in the environment.

Blender

For generating the 3D models, we will be using Blender. Blender is a useful 3D modelling tool which allows developers to create immersive and realistic 3D objects in Unity or other game engines. The simplicity of blender and ability to create rigid 3D models is useful for this project as we need to create assets like slot machines for the player to interact with.

C#

The programming will be done using C#. C# and unity allows players to generate scripts for individual objects which will allow the game environment to behave more as a game rather than a scene. Unity itself is just used to render everything so any sort of logic which is needed will be done in C# by generating scripts for individual objects within the scene and telling C# how to alter the environment. Mainly for this prototype, C# is used for scripting to set the backend logic for the slot machine. When the player interacts with the slot machine, scripting in the backend works out the logic to present visuals in the virtual environment.

Virtual Reality Environment

Core:

At the core of the interaction system are the Player, Hand and Interactable classes. The provided Player prefab sets up the player object and the SteamVR camera for the scene.

- **The interaction system works by sending messages to any object that the hands interact with. These objects then react to the messages and can attach themselves to the hands if wanted.**
- **To make any object receive messages from the hands just add the Interactable component to that object. This object will then be considered when the hand does its hovering checks.**
- **We have also included a few commonly used interactables such as the Throwable or the LinearDrive.**
- **The Player prefab also creates an InputModule which allows the hands to mimic mouse events to easily work with Unity UI widgets.**
- **The interaction system also includes a fallback mode which allows for typical first-person camera controls using the keyboard and mouse. This also allows the mouse to act like one of the player's hands. This mode is particularly useful when not everyone on the team has access to VR headset.**

Player

The Player prefab is at the core of the entire system. Most of the other components depend on the player to be present in the scene.

- The player itself doesn't do much except for keep track of the hands and the hmd.
- It can be accessed globally throughout the project and many aspects of the interaction system assume that the Player object always exists in the scene.
- It also keeps track of whether you are in VR mode or 2D fallback mode.
- Using the accessors through the Player class allows the other components to function similarly without knowing if the VR headset or mouse/keyboard is being used.
- The 2D fallback mode is useful but has its limitations. We mainly used this mode for testing out very simple interactions that only required 1 hand and the trigger button. It was mainly useful during development when not everyone on the team had a VR headset on controllers with them at all times.
- The Player also includes a few useful properties:
    - hmdTransform: This will always return the transform of the current camera. This could be the VR headset or the 2D fallback camera.

    - feetPositionGuess: This guesses the position of the player's feet based on the where the hmd is. Since we don't actually know the position of their feet, this can be pretty inaccurate depending on how the player is standing.
    - bodyDirectionGuess: This is similar to the feetPositionGuess in that it can be inaccurate depending on how the player is standing.

Teleport

- Within the environment, the main mode of moving around is using teleportation. This method is used to accommodate room restriction. In Unity a teleportation Library is used from Steam. Teleportation in the virtual reality environment is done by holding the VR controller and releases to the teleport point.
- The teleport system supports teleporting to specific teleport points or a more general teleport area.
- The important classes are Teleport, TeleportPoint and TeleportArea.
- All the functionality is wrapped up in the Teleporting prefab in Teleport/Prefabs. This prefab includes all the logic for the teleport system to function.
- Add TeleportPoints or TeleportAreas to the scene to add spots where the player can teleport to

<u>The Teleport Module is made of two components :</u>

- Teleport object: Provides the ability to teleport, coded in C# from Steam VR
- TeleportPoint:
    - This is a teleport point that the player can teleport to.
    - When teleporting onto these, the player will teleport at the origin of the point regardless of where on the point they were pointing.
    - These points can be named
    - The points also have the ability to teleport players to new scenes. (This isn't fully functional since you will have to hook it up to your scene loading system.)
- 7 points were used in this prototype in preparation for the final environment

Interactable

- The Interactable class is more of an identifier. It identifies to the Hand that this object is interactable.
- Any object with this component will receive the relevant messages from the Hand.
- Using just these 3 components you should be able to create many different and complex interactive objects.
- A few commonly used interactable system components that show how the system can combine these basic mechanics to create more complicated objects have been provided:

Throwable

- This is one of the most basic interactive objects.
- The player can pick up this object when a hand hovers over it and presses one of the grab buttons (usually trigger or grip).
- The object gets attached to the hand and is held there while the button is pressed.
- When the button is released then any velocity that was in the hand is given to throw object.
- This lets you create basic objects that can be picked up and thrown.

Slot Machine

- GameObject that holds the lever and the 4 reels

Lever

- GameObject for the lever. Has a few scripts attached to it for interaction and triggering reel spins and also handling different outcome scenarios.

Canvas_Reel

- A canvas drawn in 3d space. This canvas object allows us to render images onto each individual Reel.

Reel

- Image gameObjects which are place holders for the actual images that we loop through when reels are spun. These gameObjects also show the final image for each reel.

Render Model

Unlike the SteamVR_Render_Model component this Render_Model component inside the Interaction System handles both controller models as well as hand models and enabling/disabling them individually.

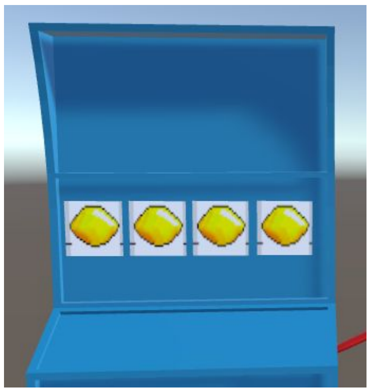
Hints

- The hint system shows hints on the controllers.
- The hints are set up in a way where each button on the controller can be called out separately.
- There is also the ability to show text hints associated with each button.

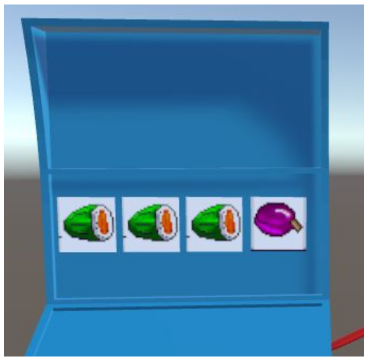
Backend Scripting

The Backend of Code of this program is done in C#. This deals with the scripting for the Reels and Lever Behaviour. This Slot Machine has three possible outcomes: **Win, Lose, and NearMisses.**
For functionality it uses the following logic:

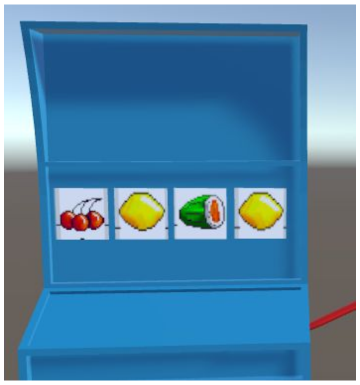
- **Win**: Reel.A.Image = Reel.B.Image=Reel.C.Image=Reel.D.Image



- **NearMisses**: Reel.1.Image = Reel.2.Image=Reel.3.Image=\=(Reel.4.Image)



- **Loses**: Any scenario that is not a Win or NearMiss.



GameObject: **Lever**

Script: **Lever_Trigger**

The Lever is used as a trigger to start spinning the reels in the slot machine. The script initializes a List of **Outcomes**(which will later be moved to an admin portal). A random element in the list is selected and used as a “predetermined” outcome for the current instance of play(reel spin).
This selected element is also removed from the **Outcomes** list, this allows us to keep track of the number of **tries** a player has attempted.
Using this strategy allows an Admin to set a predetermined list of a certain number of wins, losses and near misses and also allows to limit the number of tries the player gets.

- Enum: **OutcomeType** for the 3 different OutcomeTypes (Win, Lose, NearMisses)
- For the purpose of this prototype, there is a hardcoded list of type **OutcomeTypes** with a size of 3 set outcomes (1 Loss, 1 NearMisses and 1 Win) to demonstrate each possible outcome.
- Method: **ActivateLever**
  - Uses the steam VR **Throwable** script to check if the VR hand is attached to the lever object, if so, it initiates the reel spin process(if all conditions are met).
  - A check is done to see if the player has **tries** left, and also check that the reels are currently not in play
  - Reel scripts are activated based on **OutcomeType**
  - The **Reel_Primary_Spin** script which is attached to **ReelA** gameObject, is used to determines the results of the remaining reels
  - When reels are spinning, we set **spinReels** and **isInPlay** flags to true, and then later to false once reels are done spinning.

GameObject: **ReelA, ReelB, ReelC, ReelD**

Reel scripts are activated based on **OutcomeType** of the current play

- ReelA uses **Reel_Primary_Spin** script, which randomizes an image. This particular image is used, along with the **OutcomeType,** to determine the final images for ReelB, ReelC, ReelD
- Thus ReelB, ReelC, ReelD are spun and then the resulting image is decided based on the switch case in **Lever_Trigger** and the image of ReelA.
- Uses Randomization for simulation of spinning. An image is shown randomly once per frame. This gives an illusion of a shuffle.
- The 4 images used are:
- C# IEnumerator interface and Unity’s Coroutines functions are used to keep track of elapsed time. This allows us to end the reel spin of each reel at different times, which gives an illusion of a slow-down effect. (Each reel spin ends 2 seconds after the previous one).

GameObject: **ReelA**
Script: **Reel_ Primary_Spin**

This script is used to determine the final image that ReelA will show after randomizing the images.

- ○ To create the simulation of spinning within the virtual environment, images (from a set in unity) are randomly called for each frame
- ○ Once called it randomizes the images for the ReelA, based on images set on unity
- ○ A Sprite object, **finalSprite,** is used as a reference for **Reel_ Secondary_Spin** to set the final images of ReelB, ReelC, ReelD
- ○ Reel_Primary_Spin is initially disabled, and is later enabled and disabled in the lever_trigger script.
- ○ When this script is enabled, ReelA starts spinning until **spinReel** flag is set to false

GameObject: **ReelB, ReelC, ReelD**
Script: **Reel_ Secondary_Spin**

This script is used to set the images of the remaining reels, after the primary is called, according to its preset outcome.

- ○ To create the simulation of spinning within the virtual environment, images (from a set in unity) are randomly called for each frame
- ○ After spinning the reels, the appropriate method for the preset outcome is called(setWinImage, setLossImage, setNearMissImage), which sets an image based off the **finalSprite** from **Reel_ Primary_Spin**
- ○ Reel_**Secondary_Spin** is initially disabled, and is later enabled and disabled in the lever_trigger script.
- ○ When this script is enabled, ReelB, ReelC, ReelD start spinning until **spinReel** flag is set to false
- ○ **SetWinImage():** Used to set the current Reel's final image to ReelA's final image.
- ○ **SetLossImage():** Used to set the current Reel's final image to ReelA's any outcome except ReelA's final image.
- ○ **SetNearMissImage():** Used to set the current ReelD's final image to the image right after ReelA's final image(e.g : List[reelA_Image + 1])

References

A Psychophysiological and Behavioural Study of Slot Machine Near-Misses Using Immersive Virtual Reality
35 (3): 929–44.

Full text link: https://link.springer.com/article/10.1007/s10899-018-09822-z

“Risk Aversion and Incentive Effects.” American Economic
Review 92 (5): 1644–55.

http://content.ebscohost.com/ContentServer.asp?EbscoContent=dGJyMNHr7ESep7M4wtvhOLCmr1Gep7ZSsq64TbeWxWXS&ContentCustomer=dGJyMPGptE%2ByrLBQuePfgeyx9Yvf5ucA&T=P&P=AN&S=R&D=eoh&K=0636444

Reel Images assets: https://www.spriters-resource.com/pc_computer/klikplay/sheet/60755/

Interaction System from The Lab

Full text link:
https://valvesoftware.github.io/steamvr_unity_plugin/articles/Interaction-System.html#player